Assignment Day-8

**Assignment 1: Write a SELECT query to retrieve all columns from a 'customers' table, and modify it to return only the customer name and email address for customers in a specific city.**

let's start with a basic SELECT query to retrieve all columns from the customers table. Then, we will modify it to retrieve only the customer name and email address for customers in a specific city.

**Step 1:** Basic SELECT Query to Retrieve All Columns

SELECT \* FROM customers;

**Step 2**: Modified Query to Retrieve Customer Name and Email Address for Customers in a Specific City

To filter customers by a specific city, you need to add a WHERE clause. Assuming the columns for the customer name, email address, and city are named customer\_name, email, and city respectively, the query would look like this:

SELECT customer\_name, email

FROM customers

WHERE city = 'SpecificCity';

Replace 'SpecificCity' with the actual city you want to filter by.

**Example**

If you wanted to retrieve the customer name and email address for customers in "New York," the query would be:

SELECT customer\_name, email

FROM customers

WHERE city = 'New York';

This query will return only the customer\_name and email columns for customers who reside in "New York".

Assignment 2: Craft a query using an INNER JOIN to combine 'orders' and 'customers' tables for customers in a specified region, and a LEFT JOIN to display all customers including those without orders.

Create two queries to address the requirements:

**INNER JOIN** to combine orders and customers tables for customers in a specified region.

**LEFT JOIN** to display all customers including those without orders.

**Step 1**: INNER JOIN Query for Customers in a Specified Region

Assuming the orders table has a customer\_id column to link it to the customers table, and the customers table has a region column to specify the customer's region, the query would look like this:

SELECT customers.customer\_id, customers.customer\_name, customers.region, orders.order\_id, orders.order\_date, orders.total\_amount

FROM customers

INNER JOIN orders ON customers.customer\_id = orders.customer\_id

WHERE customers.region = 'SpecifiedRegion';

Replace 'SpecifiedRegion' with the actual region you want to filter by.

**Example**

If you wanted to retrieve the data for customers in the "West" region, the query would be:

SELECT customers.customer\_id, customers.customer\_name, customers.region, orders.order\_id, orders.order\_date, orders.total\_amount

FROM customers

INNER JOIN orders ON customers.customer\_id = orders.customer\_id

WHERE customers.region = 'West';

**Step 2**: LEFT JOIN Query to Display All Customers Including Those Without Orders

This query ensures that all customers are displayed, including those who do not have any orders. The LEFT JOIN will include all records from the customers table and match records from the orders table where available.

SELECT customers.customer\_id, customers.customer\_name, customers.region, orders.order\_id, orders.order\_date, orders.total\_amount

FROM customers

LEFT JOIN orders ON customers.customer\_id = orders.customer\_id;

**Example**

This query will list all customers and their corresponding orders, if any. Customers without orders will have NULL values in the order-related columns:

SELECT customers.customer\_id, customers.customer\_name, customers.region, orders.order\_id, orders.order\_date, orders.total\_amount

FROM customers

LEFT JOIN orders ON customers.customer\_id = orders.customer\_id;

These queries will fulfill the requirements to show specific customers and their orders as well as to display all customers including those without any orders.

**Assignment 3: Utilize a subquery to find customers who have placed orders above the average order value, and write a UNION query to combine two SELECT statements with the same number of columns.**

**Part 1: Utilizing a Subquery to Find Customers Who Have Placed Orders Above the Average Order Value**

First, let's break this down into steps:

1.Calculate the average order value.

2.Use this average to find customers who have placed orders above this value.

Assuming the orders table has order\_id, customer\_id, and total\_amount columns, the queries would be:

**Step 1:** Calculate the Average Order Value

SELECT AVG(total\_amount) AS avg\_order\_value

FROM orders;

**Step 2:** Find Customers with Orders Above the Average Order Value

Now, we'll use a subquery to incorporate this average order value into our main query.

SELECT customers.customer\_id, customers.customer\_name, orders.order\_id, orders.total\_amount

FROM customers

INNER JOIN orders ON customers.customer\_id = orders.customer\_id

WHERE orders.total\_amount > (

SELECT AVG(total\_amount)

FROM orders

);

This query finds customers who have placed orders with an amount greater than the average order value.

**Part 2: UNION Query to Combine Two SELECT Statements with the Same Number of Columns**

Let's assume we have two SELECT statements that fetch customer details from two different tables, say customers and old\_customers. Both tables have the same structure with columns like customer\_id, customer\_name, and email.

Here are two example SELECT statements:

**First SELECT Statement (from customers table):**

SELECT customer\_id, customer\_name, email

FROM customers

WHERE city = 'New York';

**Second SELECT Statement (from old\_customers table):**

SELECT customer\_id, customer\_name, email

FROM old\_customers

WHERE city = 'New York';

**UNION Query to Combine Both SELECT Statements:**

SELECT customer\_id, customer\_name, email

FROM customers

WHERE city = 'New York'

UNION

SELECT customer\_id, customer\_name, email

FROM old\_customers

WHERE city = 'New York';

The UNION operator combines the results of the two SELECT statements into a single result set, removing any duplicate rows. If you want to include duplicates, you can use UNION ALL instead.

SELECT customer\_id, customer\_name, email

FROM customers

WHERE city = 'New York'

UNION ALL

SELECT customer\_id, customer\_name, email

FROM old\_customers

WHERE city = 'New York';

These queries address both parts of the assignment: finding customers with orders above the average value using a subquery, and combining results from two tables using the UNION operator.

**Assignment 4: Compose SQL statements to BEGIN a transaction, INSERT a new record into the 'orders' table, COMMIT the transaction, then UPDATE the 'products' table, and ROLLBACK the transaction.**

the SQL statements to handle transactions involving the orders and products tables:

**Step 1:** Begin a Transaction

First, we start the transaction:

BEGIN TRANSACTION;

**Step 2**: Insert a New Record into the 'orders' Table

Assuming the orders table has columns such as order\_id, customer\_id, order\_date, and total\_amount, you would insert a new record as follows:

INSERT INTO orders (order\_id, customer\_id, order\_date, total\_amount)

VALUES (1, 123, '2024-05-21', 250.00);

**Step 3:** Commit the Transaction

If the insertion is successful and you want to commit this transaction, use:

COMMIT;

**Step 4**: Update the 'products' Table

Assuming the products table has columns such as product\_id, stock\_quantity, and you want to update the stock\_quantity of a product, you would start a new transaction and then update:

BEGIN TRANSACTION;

UPDATE products

SET stock\_quantity = stock\_quantity - 10

WHERE product\_id = 456;

**Step 5:** Rollback the Transaction

If you decide to cancel the changes made in the current transaction, use:

ROLLBACK;

**Complete Example**

Here's a complete example that incorporates all the steps:

-- Step 1: Begin a transaction

BEGIN TRANSACTION;

-- Step 2: Insert a new record into the 'orders' table

INSERT INTO orders (order\_id, customer\_id, order\_date, total\_amount)

VALUES (1, 123, '2024-05-21', 250.00);

-- Step 3: Commit the transaction

COMMIT;

-- Step 4: Begin a new transaction and update the 'products' table

BEGIN TRANSACTION;

UPDATE products

SET stock\_quantity = stock\_quantity - 10

WHERE product\_id = 456;

-- Step 5: Rollback the transaction

ROLLBACK;

**In this sequence:**

The first transaction inserts a new order and commits it.

The second transaction attempts to update the product's stock but then rolls back, undoing the update.

Make sure to adjust the column names and values to match the actual schema of your orders and products tables.

**Assignment 5: Begin a transaction, perform a series of INSERTs into 'orders', setting a SAVEPOINT after each, rollback to the second SAVEPOINT, and COMMIT the overall transaction.**

The SQL statements to handle a series of INSERT operations into the orders table, setting a SAVEPOINT after each INSERT, rolling back to the second SAVEPOINT, and finally committing the overall transaction.

**Step-by-Step SQL Statements:**

1.Begin the Transaction

2.Perform a Series of INSERTs and Set SAVEPOINTs

3.Rollback to the Second SAVEPOINT

4.Commit the Overall Transaction

Assuming the orders table has columns order\_id, customer\_id, order\_date, and total\_amount, here is how you would do it:

-- Step 1: Begin the transaction

BEGIN TRANSACTION;

-- Step 2: Perform INSERTs and set SAVEPOINTs

INSERT INTO orders (order\_id, customer\_id, order\_date, total\_amount)

VALUES (1, 123, '2024-05-21', 250.00);

SAVEPOINT savepoint1;

INSERT INTO orders (order\_id, customer\_id, order\_date, total\_amount)

VALUES (2, 124, '2024-05-22', 300.00);

SAVEPOINT savepoint2;

INSERT INTO orders (order\_id, customer\_id, order\_date, total\_amount)

VALUES (3, 125, '2024-05-23', 350.00);

SAVEPOINT savepoint3;

-- Step 3: Rollback to the second SAVEPOINT

ROLLBACK TO savepoint2;

-- Step 4: Commit the overall transaction

COMMIT;

**Assignment 6: Draft a brief report on the use of transaction logs for data recovery and create a hypothetical scenario where a transaction log is instrumental in data recovery after an unexpected shutdown.**

**Report on the Use of Transaction Logs for Data Recovery:-**

**Introduction**

Transaction logs are a crucial component of database management systems (DBMS), providing a reliable mechanism for ensuring data integrity and aiding in data recovery. They record all changes made to the database, including data modifications and structural changes, in a sequential manner. This allows the DBMS to track every transaction, making it possible to recover the database to a consistent state in the event of failures, such as unexpected shutdowns, system crashes, or other catastrophic events.

**Purpose of Transaction Logs:-**

**1.Data Integrity:** Transaction logs help maintain the integrity of the database by ensuring that all transactions are either fully completed or fully rolled back.

**2.Recovery:** In case of a system failure, transaction logs can be used to recover lost or corrupted data, ensuring minimal data loss.

**3.Auditing:** Logs provide a historical record of all transactions, which can be useful for auditing purposes and tracking changes over time.

**Mechanism of Transaction Logs**

Transaction logs typically store the following information:

* **Transaction ID:** Unique identifier for each transaction.
* **Timestamp:** Time at which the transaction occurred.
* **Before and After Images:** The state of the data before and after the transaction.
* **Type of Operation:** Insert, update, delete, etc.
* **Commit/Rollback Information**: Indicates whether the transaction was committed or rolled back.

**Data Recovery Process**

**1.Redo Operations**: Transactions that were committed before the failure are re-applied to ensure that all committed changes are present.

**2.Undo Operations:** Transactions that were in progress or rolled back at the time of the failure are undone to maintain data consistency.

**Hypothetical Scenario: Transaction Log in Data Recovery**

**Scenario Description**

Imagine a retail company that uses an online ordering system. The orders table in their database tracks all customer purchases. One day, the system experiences an unexpected shutdown due to a power outage, just as several transactions are being processed.

**Before the Shutdown**

* Transaction 1: Customer A places an order.
* Transaction 2: Customer B places an order.
* Transaction 3: Customer C starts placing an order, but the system shuts down before the transaction completes.

**Transaction Log Entries**

Before the shutdown, the transaction log records the following:

**Transaction 1:**

* Transaction ID: T1
* Operation: Insert
* Data: {OrderID: 101, CustomerID: A, TotalAmount: $50}
* Status: Committed

**Transaction 2:**

* Transaction ID: T2
* Operation: Insert
* Data: {OrderID: 102, CustomerID: B, TotalAmount: $75}
* Status: Committed

**Transaction 3:**

* Transaction ID: T3
* Operation: Insert
* Data: {OrderID: 103, CustomerID: C, TotalAmount: $60}
* Status: In Progress

**Recovery Process**

After the system is restored, the DBMS uses the transaction log to recover the database:

**Redo Committed Transactions:** Transactions T1 and T2 are re-applied to ensure the orders from Customer A and Customer B are present in the orders table.

**Undo In-Progress Transactions:** Transaction T3 is rolled back, ensuring that the incomplete order from Customer C does not corrupt the database.

**Post-Recovery State**

The orders table correctly reflects the orders from Customer A and Customer B.

There is no partial or corrupt entry for Customer C’s order, maintaining the integrity of the database.

**Conclusion**

Transaction logs are vital for maintaining data integrity and facilitating data recovery in database systems. By recording every transaction, they ensure that in the event of an unexpected shutdown or system failure, the database can be restored to a consistent and accurate state, minimizing data loss and corruption.